# Python Beginner's Cheat Sheet

**Variables**
Variables are placeholders for pieces of information that can change. They store data and allow the program to retrieve the stored data by calling its name.

**Naming and printing a variable**

```
msg = “hello world!”
print(msg)

>>> hello world!
```

The message “hello world” is placed inside a variable named msg
The print() function will display the message when the user runs the program

**Rules for naming variables**
1. Start with a letter or underscore
2. Consists of letter, number, or underscore
3. Case-sensitive
4. Cannot be a reserved word

**Reserved words in Python**

| | | |
|---|---|---|
| False | elif | lambda |
| True | else | nonlocal |
| None | except | not |
| and | finally | or |
| As | for | pass |
| assert | from | raise |
| break | global | return |
| continue | if | try |
| class | import | while |
| def | in | with |
| del | is | yield |

**Expressions**
Expressions are a combination of variables and operators, with or without literals, that evaluates to a value.

**Types of literals in Python**

String literals
A string literal can be created by writing a text or a group of characters surrounded by single (‘ ’), double (“ ”), or triple quotes (“”” “”” / ‘’’ ‘’’)

Numeric literals
Numeric literals are immutable and have three main categories - Integer, Float, and Complex.
- Integers include both positive and negative whole numbers, including zero.
- Floats are real numbers having both integer and fractional parts.
- Complex numerals will be in the form a + bj, where ‘a’ is the real part and ‘b’ is the complex part.

Boolean literals
There are only two boolean literals in Python - True and False
- True represents the value as 1
- False represents the value as 0

Literal collections
In Python, there are four different types of literal collections - list, tuple, dictionary, and set.

Special literals
There is one special literal in Python. “None” is used to define a null variable. If “None” is compared with anything else other than another “None”, it will return False.

**Statements**
Statements are an instruction that the Python interpreter can execute.

**Examples of statements**
- if statement
- assignment statement
- for statement
- while statement

**Multi-line statement**
In Python, a statement can be extended over multiple lines with the line continuation character (\).

```
a = 1 + 2 + 3 + \
    4 + 5 + 6 + \
    7 + 8 + 9
```

The above example is an explicit line continuation. In Python, line continuation is implied inside round brackets, square brackets, and braces.

```
colors = ['red',
          'blue',
          'green']
```

We can also put multiple statements in a single line using semicolons.

```
a = 1; b = 2; c = 3
```

**Data Types in Python**
Python has the following data types built-in by default, in these categories:

- Text type (str)
- Numeric types (int, float, complex)
- Sequence types (list, range, tuple)
- Mapping types (dict)
- Set types (set, frozenset)
- Boolean types (bool)
- Binary types (bytes, bytearray, memoryview)

The type() function will let you get the data type of any object.

**Setting the data type**
Python automatically sets the data type when you assign a value to a variable.

**Setting the specific data type**
If you want to specify the data type, use constructor functions.

```
x = str(“hello world”)
x = int(20)
```

```
x = float(20.5)
x = complex(1j)
x = list(("apple", "banana", "cherry"))
x = tuple(("apple", "banana", "cherry"))
x = range(6)
x = dict(name="John", age=36)
x = set(("apple", "banana", "cherry"))
x = frozenset(("apple", "banana", "cherry"))
x = bool(5)
x = bytes(5)
x = bytearray(5)
x = memoryview(bytes(5))
```

**Mixing data types**

1. Operations between integer and floating point number yields floating point number
2. A string multiplied with an integer returns a repeated string value
3. Division of two numbers results in a floating point number
4. A number and a string cannot be added together
5. A float and a string cannot be added together
6. A boolean and a string cannot be added together

**Operators and operands**

Operators are special symbols that represent computations (e.g. addition and subtraction). The values the operator uses are called operands.

Python divides the operators into the following groups:

1. Arithmetic operators
2. Assignment operators
3. Comparison operators
4. Logical operators
5. Identity operators
6. Membership operators
7. Bitwise operators

**Arithmetic operators**

Arithmetic operators are used with numeric values to perform common mathematical operations.

| Name | Operator | Example |
|---|---|---|
| Addition | + | x + y |
| Subtraction | - | x - y |
| Multiplication | * | x * y |
| Division | / | x / y |
| Floor division | // | x // y |
| Exponentiation | ** | x ** y |
| Modulo | % | x %  y |

**Assignment operators**
Assignment operators are used to assign values to variables.

| Operator | Example | Same As |
|---|---|---|
| = | x = 5 | x = 5 |
| += | x += 5 | x = x + 3 |
| -= | x -= 5 | x = x - 3 |
| *= | x *= 5 | x = x * 3 |
| /= | x /= 5 | x = x / 3 |
| %= | x %= 5 | x = x % 3 |
| //= | x //= 5 | x = x // 3 |
| **= | x **= 5 | x = x ** 3 |
| &= | x &= 3 | x = x & 3 |
| \|= | x \|= 3 | x = x \| 3 |
| ^= | x ^= 3 | x = x ^ 3 |
| >>= | x >>= 3 | x = x >> 3 |
| <<= | x <<= 3 | x = x << 3 |

**Comparison operators**
Comparison operators are used to compare two values.

| Operator | Name | Example |
|---|---|---|
| Equal to | == | x == y |
| Not equal to | != | x != y |
| Greater than | > | x > y |
| Less than | < | x < y |
| Greater than or equal to | >= | x >= y |
| Less than or equal to | <= | x <= y |

## Logical operators

Logical operators are used to combine conditional statements.

| Operator | Description | Example |
|---|---|---|
| and | Returns true if both statements are true | x < 5 and x < 10 |
| or | Returns true if one of the statements is true | x < 5 or x < 4 |
| not | Reverses the result of the condition in the brackets | not (x < 5 and x < 10) |

## Identity operators

Identity operators are used to determine if two objects are the same object and have the same memory location.

| Operator | Description | Example |
|---|---|---|
| is | Returns true if both variables are the same object | x is y |
| is not | Returns true if both variables are not the same object | x is not y |

## Membership operators

Membership operators are used to test if a sequence is presented in an object.

| Operator | Description | Example |
|---|---|---|
| in | Returns true if both variables are the same object | x is y |
| not in | Returns true if both variables are not the same object | x is not y |

**Bitwise operators**

Bitwise operators are used to compare binary numbers.

| Operator | Name | Description |
|---|---|---|
| & | AND | Sets each bit to 1 if both bits are 1 |
| \| | OR | Sets each bit to 1 if one of two bits is 1 |
| ^ | XOR | Sets each bit to one if only one of two bits is 1 |
| ~ | NOT | Inverts all bits |
| << | Zero fill left shift | Shift left by pushing zeros in from the right, and let the leftmost bits fall off |
| >> | Signed right shift | Shift right by pushing zeros in from the left, and let the rightmost bits fall off |

**Order of operations**

When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence. The acronym PEMDAS is a useful way to remember Python's order of operations.

- Parentheses (highest)
- Exponentiation
- Multiplication
- Division
- Addition
- Subtraction (lowest)

**Input using input() function**

The input() function allows the program to read from the keyboard. When called, the program flow stops until the user gives an input and ends it by pressing the enter key. The user's input will always be returned as a string.

```
num = int(input("enter a number: "))
```

**Output using output() function**

The print() function takes in any number of parameters and prints them out on one line of text.

**Output formatting**

There are several ways to format an output in Python.

1. String concatenation operator (+)

```
food = pizza
print(“hello is your favourite food ” + food + “ ?”)
```

2. String modulo operator (%)

```
name = sam
weight = 60.5
print(“hello  %s you are %0.2f kg” (name, weight))
```

3. String method “format” (f)

```
admin _no = 285720J
gender = male
print(f”your admin number is {admin_no} and you are {gender}”)
```

4. string.format() method

```
x = 5; y = 10
print('The value of x is {} and y is {}'.format(x,y))
```

**Built-in exceptions in Python**

| Exception | Description |
|---|---|
| AssertionError | Raised when the assert statement fails |
| AttributeError | Raised on the attribute assignment or reference fails |
| EOFError | Raised when the input() function hits the end-of-file condition |
| FloatingPointError | Raised when a floating point operation fails |
| GeneratorExit | Raised when a generator's close() method is called |
| ImportError | Raised when the imported module is not found |

| | |
|---|---|
| IndexError | Raised when the index of a sequence is out of range |
| KeyError | Raised when a key is not found in a dictionary |
| KeyboardInterrupt | Raised when the user hits the interrupt key |
| MemoryError | Raised when an operation runs out of memory |
| NameError | Raised when a variable is not found in the local or global scope |
| NotImplementedError | Raised by abstract methods |
| OSError | Raised when a system operation causes a system-related error |
| OverflowError | Raised when the result of an arithmetic operation is too large to be represented |
| ReferenceError | Raised when a weak reference proxy is used to access a garbage collected referent |
| RuntimeError | Raised when an error does not fall under any other category |
| StopIteration | Raised by the next() function to indicate that there is no further item to be returned by the iterator |
| SyntaxError | Raised by the parser when a syntax error is encountered |
| IndentationError | Raised when there is an incorrect indentation |
| TabError | Raised when the indentation consists of inconsistent tabs and spaces |
| SystemError | Raised when the interpreter detects an internal error |
| SystemExit | Raised by the sys.exit() function |

| | |
|---|---|
| TypeError | Raised when a function or operation is applied to an object of an incorrect type |
| UnboundLocalError | Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable |
| UnicodeError | Raised when a Unicode-related encoding or decoding error occurs |
| UnicodeEncodeError | Raised when a Unicode-related error occurs during encoding |
| UnicodeDecodeError | Raised when a Unicode-related error occurs during decoding |
| UniTranslateError | Raised when a Unicode-related error occurs during translation |
| ValueError | Raised when a function gets an argument of correct type but of improper value |
| ZeroDivisionError | Raised when the second operand of a division or modulo operation is zero |

**Error handling with try-except block**
The try block contains the code to be monitored for the exceptions.
The except block contains what is to be done if a specific exception occurs.
The else block is executed only if no exceptions occur in the try block.
The finally block is used for clean up code and is always executed.
The else and finally blocks are optional.

```
try:
        f = open("demofile.txt")
        try:
                f.write("lorum Ipsum")
        except:
                print("Something went wrong when writing to the file")
        finally:
                f.close()
except:
        print("Something went wrong when opening the file")
```

**Raise an exception**
You can choose to throw an exception if a condition occurs. To throw (or raise) an exception, use the raise keyword.

```
x = -1
if x < 0:
        raise Exception(“sorry, no numbers below zero”)
```

You can also define what kind of error to raise, and the text to print to the user.

```
x = “hello”
if not type(x) is int:
        raise TypeError(“only integers allowed”)
```

**Flowcharts**
A graphical representation of the program steps. Contains different types of symbols to represent different types of operations. The symbols are connected by arrows that represent the flow of the program.

**Basic symbols used in flowchart designs**

Terminal
The oval symbol indicates start, stop, and pause in a program’s logic flow. A pause is generally used in a program logic under some error conditions. The terminal is the first and last symbol in the flowchart.

Input/Output
A parallelogram denotes any function of input/output type. Program instructions take input from input devices and display output on output devices are indicated with parallelogram in a flowchart.

Processing
A box represents arithmetic instructions. All arithmetic processes such as adding, subtracting, multiplication, and division are indicated by action or process symbols.

Decision
The diamond symbol represents a decision point. Decision-based operations such as yes/no question or true/false are indicated by diamond in the flowchart.

Connectors
Whenever the flowchart becomes complex, or it spreads over more than one page, it is useful to use connectors to avoid any confusion. It is represented by a circle.

Flow lines
Flow lines indicate the exact sequence in which instructions are executed. Arrows represent the direction of flow of control and the relationship among different symbols of the flowchart.

**if-elif-else statement**
if statements are used when we want to execute a code block only if a given test condition is True.

elif statements are used when we want to check for multiple conditions. If the first elif condition is False, the next elif test condition is checked.

else statements are executed when if and elif statements above it are all False.

```
if age < 4:
        ticket_price = 5
elif age < 18:
        ticket_price = 10
else:
        ticket_price = 15
```

**Nested if statement**
Nested if statements are used when you want to test conditions inside a condition.

```
num = 21
if num < 25:
        if num % 2 == 0:
                print(“num is even and less than 25”)
        else:
                print(“num is odd and less than 25”)
else:
        print(“num is greater than 25”)
```

**Checking multiple conditions in one line**
Multiple conditions can be checked at the same time.

The and operator returns true if all the conditions listed are True.

```
age1 = 22
age 2 = 18

if age1 >= 20 and age2 >= 20:
>>> False

if age1 >= 15 and age2 >= 15:
>>> True
```

The or operator returns True if any one condition is True.

```
age1 = 22
age 2 = 18

if age1 >= 20 or age2 >= 20:
>>> True

if age1 >= 25 or age2 >= 25:
>>> False
```

**for loops**
for loops are count-controlled loops, where the exact number of times the loop will be executed is known.

```
numbers = [1, 2, 3, 4, 5,  6, 7, 8, 9]
for i in numbers:
        print(i)
```

**while loops**
while loops are condition-controlled loops, where it is hard to tell when it will terminate. A while loop will execute until its condition becomes False.

```
value = 1
while value <= 5:
        print(value)
        value += 1
```

**Infinite loops**
Every while loop needs a way to stop running else it will continue to run forever. If the condition never becomes false, the loop will never stop running.

This while loop will print “nice to meet you john” for an infinite number of times because the loop condition never becomes false.

```
while True:
        name = john
        print(“nice to meet you john”)
```

**break statements**
break statements and used when we want to exit a loop completely and jump to the statement immediately following it.

```
city = input(“what cities have you visited, enter quit when you’re done”)
while True:
        if city == “quit”:
                break
        else:
                print(f“I have been to {city}”)
```

**continue statements**
continue statements are used when we want to skip a part of a loop and start the next iteration.

```
value = 10
while value > 0:
        value -= 1
        if value == 5:
                continue
        print(f“current value: {value}”)
print(“end of program”)
```

**Lists**
Lists are an object that contains multiple data items of any type. They are mutable, and its contents can be manipulated. Lists are surrounded by square brackets and elements are separated by commas.

**Accessing list elements**
Individual elements in a list are accessed according to their index position. The index of the first element is 0, the index of the second element is 1, and so on. Negative indices refer to items at the end of the list.

To access a particular element, write the name of the list followed by the index of the element in square brackets.

```
first_element = myList[0]
second_element = myList[1]
third_element = myList[2]
```

**Modifying list elements**
To modify an element in a list, refer to the index of the item you want to modify.

```
myList = [“apple”, “banana”, “cherry”]

myList[0] = “orange”
>>>[“orange”, “banana”, “cherry”]
```

**Adding elements to list**
To add elements to the end of the list, use the append() function.
To add elements at a particular position, use the insert() function.

```
myList = [“john”, “bob”, “charlie”]

myList.append(“jane”)
>>> myList = [“john”, “bob”, “charlie”, “jane”]

myList.insert(3, “amy”)
>>> myList = [“john”, “bob”, “charlie”, “amy”, “jane”]
```

**Removing elements in a list**
Remove elements by their position in a list or by their value. If you remove an

item by its value, Python will only remove the first item with that value.

```
del users [-1]

myList.remove(“bread”)
```

**Popping elements of the list**
By default, the pop() function will return the last element in the list. However, you can also pop elements from any position in the list by specifying the index position.

```
myList = [1, 5, 8, 9, 10, 4]

newList = myList.pop()
>>> 4

firstElement = myList.pop(0)
>>> 1
```

**Slicing a list**
A list can be sliced into many sub-lists.

To slice a list, start with the index of the first item you want. Next, add a colon and the index after the last item you want.

To start at the beginning of the list, leave off the first index. To stop at the end of the list, leave off the last index.

Get the first three elements.

```
names = [“amy”, “beth”, “charles”, “dan”, “eric”]
firstThree = names[ : 3]
```

Get the middle three elements.

```
middleThree = names[1 : 4]
```

Get the last three elements.

```
lastThree = names[-3 : ]
```

## Dictionaries

Dictionaries are objects that store a collection of data. Each element in the dictionary has two parts - a key and a value. Dictionaries are mutable, but their contents indexed by keys are immutable. Dictionaries are surrounded by curly braces and their key-value pairs are separated by commas.

## Accessing dictionary values

To access the values associated with an individual key, enter the name of the dictionary and then indicate the key in a set of square brackets. If a key you entered is not in the dictionary, an error will occur.

```
colors = {“first”: “red”, “second”: “black”, “third”: “blue”}

colors[“first”]
>>> red
colors[“third”]
>>> blue
```

The get() method can also be used to access values. This method will return None instead of an error message if the key entered does not exist.

```
colors = {“first”: “red”, “second”: “black”, “third”: “blue”}

color.get(“second”)
>>> black
```

## Modifying dictionary values

To modify dictionary values, enter the name of the dictionary, enclose the key in square brackets, then provide the new value for that key.

```
colors = {“first”: “red”, “second”: “black”, “third”: “blue”}

colors[“first”] = “yellow”
>>> {“first”: “yellow”, “second”: “black”, “third”: “blue”}
```

## Adding new key-value pairs

To add new key-value pairs to an existing dictionary, enter the name of the dictionary and the new key in square brackets, then set it equal to the new value.

```
colors = {“first”: “red”, “second”: “black”, “third”: “blue”}

colors[“fourth”] = “green”
>>> {“first”: “red”, “second”: “black”, “third”: “blue”, “fourth”: “green”}
```

**Removing key-value pairs**
To remove key-value pairs from a dictionary, use the del keyword and the dictionary name followed by the key in square brackets.

```
colors = {“first”: “red”, “second”: “black”, “third”: “blue”}

del colors[“first”]
>>> { “second”: “black”, “third”: “blue”}
```

**Tuples**
Tuples are objects that contain multiple data items of any type. They are just like a list, except that they are immutable and items cannot be added or removed. Tuples are surrounded by round brackets and their elements are separated by commas.

```
tuple1 = (“one”, “two”, “three”)
tuple2 = (1,  9, 10, 5)
tuple3 = (True, False, True)
```

**Functions**
Functions are a group of statements that exist within a program for the purpose of performing a specific task.  Functions provide usable codes to reduce development time.

**Components of a function**

<u>Function’s header</u>
The def keyword is used to mark the start of the function.
A name is given to the function to uniquely identify it.
Parameters are passed into the function (optional).
The colon is used to mark the end of the function header.

Function's body
Docstrings are used to describe what the function does.
Valid python statements will make up the body of the function.

Function's ending
Generally, a print and/or return statement is used at the end of a function.
A return statement is used to return a value to the function (optional).
Note! A function will stop running when it reaches a return statement.

Calling a function
To call a function, enter the name of the function followed by a set of parentheses containing arguments if any.

```
def sayHello (name):
        """
        this function will say hello to the person passed in as a
        parameter
        """
        print(f"hello {name}, how are you?")

sayHello("john")
```

**Built-in functions in Python**
The Python interpreter has a number of functions and types built into it that are always available.

| **Function** | **Definition** |
|---|---|
| abs() | Return the absolute value of a number. |
| aiter() | Return an asynchronous iterator for an asynchronous iterable. |
| all() | Return True if all elements of the *iterable* are true (or if the iterable is empty). |
| any() | Return True if any element of the *iterable* is true. If the iterable is empty, return False. |
| anext() | When awaited, return the next item from the given asynchronous iterator, or *default* if given and the iterator is exhausted. |

| | |
|---|---|
| ascii() | Returns a string containing a printable representation of an object and escapes the non-ASCII characters in the string using \x, \u, or \U escapes. |
| bin() | Convert an integer number to a binary string prefixed with "0b". |
| bool() | Return a Boolean value, i.e. one of True or False. |
| breakpoint() | A tool that allows developers to set points in code at which a debugger is called. |
| bytearray() | Return a new array of bytes. The bytearray class is a mutable sequence of integers in the range 0 <= x < 256. |
| bytes() | Return a new "bytes" object which is an immutable sequence of integers in the range 0 <= x < 256. |
| callable() | Return True if the object argument appears callable, False if not. If this returns True, it is still possible that a call fails, but if it is False, the calling object will never succeed. |
| chr() | Return the string representing a character whose Unicode code point is the integer *i*. |
| classmethod() | Transform a method into a class method. A class method receives the class as an implicit first argument, just like an instance method receives the instance. |
| compile() | Compile the *source* into a code or AST object. |
| complex() | Return a complex number with the value *real* + *imag**1j or convert a string or number to a complex number. |
| delattr() | The function deletes the named attribute, provided the object allows it. |
| dict() | This function creates a dictionary. |
| dir() | Without arguments, return the list of names in |

| | |
|---|---|
| | the current local scope. With an argument, attempt to return a list of valid attributes for that object. |
| divmod() | Take two (non-complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using integer division. |
| enumerate() | Allows you to loop over an iterable object and keep track of how many iterations have occurred. |
| eval() | Parses the expression argument and evaluates it as a python expression. |
| exec() | Used for the dynamic execution of the Python program which can either be a string or object code. |
| filter() | Allows you to process an iterable and extract those items that satisfy a given condition. |
| float() | Used to return a floating-point number from a number or a string. |
| format() | A technique of the string category that permits you to try and do variable substitutions and data formatting. |
| frozenset() | Takes an iterable object as input and makes them immutable. |
| getattr() | Used to access the attribute value of an object and also gives an option of executing the default value in case of unavailability of the key. |
| globals() | Returns the dictionary of the current global symbol table. |
| hasattr() | An inbuilt utility function, which is used to check if an object has the given named attribute and return true if present, else false. |
| hash() | Returns the hash value of an object if it has one. |

| | |
|---|---|
| help() | Used to get the documentation of specified module, class, function, variables, etc. |
| hex() | Converts the specified number into a hexadecimal value. The returned string always starts with the prefix 0x. |
| id() | Accepts a single parameter and is used to return the identity of an object. |
| input() | Allows a user to insert a value into a program and returns a string value. |
| int() | Returns an integer from a given object or converts a number in a given base to decimal. |
| isalnum() | Return True if string is alphanumeric. |
| isalpha() | Return True if string contains only letters. |
| isdigit() | Return True if string contains only digits. |
| isinstance() | Allows you to verify a particular value's data type. |
| isidentifier() | Return True if string is valid identifier. |
| issubclass() | Used to check if a class is a subclass of another class or not. |
| isspace() | Return True if string contains only white space. |
| isupper() | Returns True if all characters of the string are uppercase. |
| islower() | Returns True if all characters of the string are lowercase. |
| iter() | Returns an iterator for the given object. |
| len() | Returns the length of a list, string, dictionary, or any other iterable data format in Python. |
| list() | This function creates a list object. |
| locals() | Returns the dictionary of the current local symbol table. |

| | |
|---|---|
| lower() | Converts all characters of the string to lowercase. |
| map() | Allows you to process and transform all the items in an iterable without using an explicit for loop. |
| max() | Returns the largest item in an iterable or the largest of two or more arguments. |
| memoryview() | Allows direct read and write access to an object's byte-oriented data without needing to copy it first. |
| min() | Returns the lowest value in a list of items. |
| next() | Used to return the next item from the iterator. |
| object() | Returns an empty object. |
| oct() | Takes an integer and returns the octal representation in a string format. |
| open() | Opens a file, and returns it as a file object. |
| ord() | Converts a character into its Unicode code value. |
| pow() | Returns the power of the given numbers. |
| print() | Takes in python data and prints those values to standard out. |
| property() | Provides an interface to instance attributes. |
| range() | Returns a sequence of numbers starting from zero and increment by 1 by default and stops before the given number. |
| repr() | Returns a printable representational string of the given object. |
| reversed() | Allows us to process the items in a sequence in reverse order. |
| round() | Returns a floating point number that is a rounded version of the specified number, with |

| | |
|---|---|
| | the specified number of decimals. |
| set() | Used to convert any of the iterable to a sequence of iterable elements with distinct elements. |
| setattr() | Used to assign the object attribute its value. |
| slice() | Returns a slice object. A slice object is used to specify how to slice a sequence. |
| sorted() | Returns a sorted list from the iterable object. |
| staticmethod() | Transform a method into a static method. A static method does not receive an implicit first argument. |
| str() | Converts values to a string form so they can be combined with other strings |
| sum() | Adds up all the numerical values in an iterable and returns the total of those values. |
| super() | Lets you access methods from a parent class from within a child class. |
| tuple() | This function creates a tuple. |
| type() | Returns the data type of the object passed to it as an argument. |
| upper() | Converts all characters of the string to uppercase. |
| vars() | Returns the __dic__ attribute of an object. The __dict__ attribute is a dictionary containing the object's changeable attributes. |
| zip() | Returns a zip object. |
| __import__() | This function is not necessary for everyday Python programs. It is rarely used and often discouraged. It can be used to change the semantics of the import statement as the statement calls this function. |

**Modules**
A module is a Python file that can be imported inside another Python program. It can define functions, classes, and variables. Python has a from statement that allows the import of all names from a module into the namespace.

**Storing a function in a module (filename: pizza.py)**

```
def make_pizza(size, *toppings):
        """code to make a pizza"""
        print(f"making a {size} pizza")
        for topping in toppings:
print(f"- {topping}")
```

**Importing an entire module (filename: makingPizzas.py)**

```
import pizza

pizza.make_pizza("medium", "pepperoni")
pizza.make_pizza("large", "pineapple", "ham")
```

**Importing a specific function from a module**

```
from pizza import make_pizza

make_pizza("medium", "pepperoni")
make_pizza("large", "pineapple", "ham")
```

**Importing all functions from a module**

```
from pizza import *

make_pizza("medium", "pepperoni")
make_pizza("large", "pineapple", "ham")
```

**Giving a module an alias**

```
import pizza as p
```

```
p.make_pizza(“medium”, “pepperoni”)
p.make_pizza(“large”, “pineapple”, “ham”)
```

**Giving a module an alias**

```
from pizza import make_pizza as mp

mp.make_pizza(“medium”, “pepperoni”)
mp.make_pizza(“large”, “pineapple”, “ham”)
```

**Comments**
Comments can be used to explain Python code, make code more readable, and to prevent execution when testing code.

A comment can be written in three ways:
1. Entirely on its own line
2. Next to a statement of code
3. As a multi-line comment block

**Writing a comment entirely on its own line**

```
#defining the postal code
code = 936582
```

**Writing a comment next to a statement of code**

```
#defining the general structure of a product with default values
product = {
   "productId": 0,          # product reference id, default: 0
   "description": " ",      # item description, default: empty
   "categoryId": 0,         # item category, default: 0
   "price": 0.00            # price, default: 0.00
}
```

**Writing a comment as a multi-line block**
To add a multi-line comment, either insert a hash character at the start of each

line or use a multi-line string (triple quotes - “”” “”” / ‘’’ ‘’’).

```
# LinuxThingy version 1.6.5
# Parameters:
# -t (--text): show the text interface
# -h (--help): display this help
```

```
"""
LinuxThingy version 1.6.5
Parameters:
-t (--text): show the text interface
-h (--help): display this help
"""
```

**Useful websites to learn Python**

Udemy
Udemy is a popular online course platform, containing probably the biggest collection of online courses.
We recommend:
- Introduction to Python Programming (4.5 star ratings)
  https://www.udemy.com/course/pythonforbeginnersintro/
- Python for Absolute Beginners! (4.5 star ratings)
  https://www.udemy.com/course/free-python/
- Learn Python for Total Beginners (4.4 star ratings)
  https://www.udemy.com/course/python-3-for-total-beginners/

CodeCademy
CodeCademy is an online interactive platform that offers free coding classes in 12 different programming languages, including Python.
We recommend:
- Learn Python 2
  https://www.codecademy.com/learn/learn-python
- Learn Python 3 (only for paying users)
  https://www.codecademy.com/learn/learn-python-3

Coursera
Coursera is an online learning platform that offers thousands of online courses with over 200 of the world’s learning universities and companies.
We recommend:
- Programming for Everybody (Getting Started with Python)

https://www.coursera.org/learn/python

FreeCodeCamp
FreeCodeCamp is a non-profit organization that aims to help people learn to code for free.
We recommend:

- Learn Python Full Course for Beginners [YouTube]
  https://www.youtube.com/watch?v=rfscVS0vtbw
- Learn Python Free Python Courses for Beginners Article
  https://www.freecodecamp.org/news/learn-python-free-python-courses-for-beginners/

w3resource
w3resource offers web development tutorials and contains practice questions with solutions for many web development languages, including Python.
We recommend:

- Python Tutorial
  https://www.w3resource.com/python/python-tutorial.php
- Python Exercises, Practise, Solution
  https://www.w3resource.com/python-exercises/

W3Schools
W3Schools is a web developer's site, with tutorials and references on web development languages, including Python.
https://www.w3schools.com/python/

Stack Overflow
Stack Overflow is a community-based space to find and contribute answers to technical challenges.
https://stackoverflow.com/questions/tagged/python

Real Python
At Real Python you can learn all things Python from the ground up. Everything from the absolute basics of Python, to web development and web scraping, to data visualization, and beyond.
https://realpython.com/